INDIANA UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 389

Behavior Tables: A Basis for System Representation and Transformational System Synthesis

Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson

AUGUST 1993

To appear in the proceedings of the *1993 IEEE/ACM International Conference on Computer-Aided Design* (*ICCAD '93*), Santa Clara, California, November, 1993.

Behavior Tables: A Basis for System Representation and Transformational System Synthesis *

Kamlesh Rath†, M. Esen Tuna, and Steven D. Johnson

Computer Science Department, Indiana University
Bloomington, Indiana 47405

Abstract

In this paper, we introduce behavior tables, an extension of register transfer tables, as a unified basis for reasoning about control, datapath, protocol, and data expansion facets of system synthesis. Behavior tables can model indirection in system specification, by allowing names of registers and states to be treated as values. Behavior tables are based on a finite state machine model and provide a framework for transformational design to derive a formally correct implementation from a specification. To illustrate our approach, we sketch some transformations on a behavior table description of the FM9001 processor.

1 Introduction

Behavior tables are an extension of register transfer tables that provide a unified design representation for control, datapath, protocol and data expansion facets of system behavior. The feature that distinguishes behavior tables from other system representations and hardware description languages is that, they can model *indirection* in system description. We present a set of transformations on different facets of behavior tables which can be used to derive an implementation from a specification. Behavior tables and the transformations on them are based on a finite-state machine model. Along with the global view of a design, behavior tables can also provide abstract views of a design. A table form can be a useful visual output for a designer to interact with the design tool.

Many design automation systems use graph based structures for design representation [1, 2, 3], which are suited for either control-flow or data-flow representation. Petri-net based internal representations are also suited for control-flow representation but are not useful for data-flow representation [4]. A hierarchical design representation (e.g [5]) can not effectively represent orthogonal facets of a system. Behavior tables provide a unified representation on which transformations on different facets of a design can be performed.

Behavior tables can be decomposed into interacting sequential components using a general transformation that takes the protocol specification between components as a parameter. These components can be independently synthesized or mapped to off-the-shelf components. SpecPart [6] partitions algorithm/process grained computations from the SpecChart specifications using default protocols. The CHOP system-level design partitioner [7] uses special purpose modules on both sides of every interaction. Partitioning in the System Architect's Workbench is accomplished by behavioral transformations using a very simple message passing protocol. All these approaches are limited by the interaction protocols supported by the partitioning methods.

Indirection in system description is modeled by allowing names of registers and states to be treated as values in the system. Behavior tables also provide a mechanism to change levels of data abstraction. The research reported here grew out of our existing design derivation system, which is based on first order functional algebra [8, 9].

2 Behavior Tables

A behavior table is a representation for a finite state machine that models system behavior. Each row in a behavior table represents a transition in the machine described by the table. The columns are divided into two sections, the decision section and the action section. The decision section represents state and conditions that must hold for the transition to be executed.

*Research reported herein was supported, in part, by NSF, under grants numbered MIP 89-21842 and MIP 92-08745.

†Email: rathk@cs.indiana.edu

The action section represents data flow through the functional units, ports, and next state.

A machine $M = \langle T, s, n, P, R, C, S, \hat{I}, V \rangle$, where T is a non-empty set of transitions, s is the present state, n is the next state, P is the set of ports, R is a set of internal registers and signals, C is the set of internal predicates, S is a set of states, $\hat{I}$ is a set of reference sets, and V is the domain of values including the don't care value (#), and functions over registers/ports. The set of ports ($P = CI \cup CO \cup DI \cup DO$) is a union of the sets of control inputs, control outputs, data inputs, and data outputs. We adopt a convention that references to a register/port r is written as r̂, and the prefix @ denotes dereference.

A transition is defined as a function of the form $t = \ if\,[satisfies(t_d, \vec{d})]\ then\ t_a$, where the assignment function $t_d : C \cup CI \mapsto V$ and $t_d : \{s\} \mapsto S$. An assignment function t_d is satisfied with respect to the current conditions $\vec{d}$ if $(\forall c_i \in \{s\} \cup C \cup CI\ .\ t_d(c_i) = \vec{d_i}\ \vee\ t_d(c_i) = \#)$ is *true*. The action section of each transition is defined by the assignment function, $t_a : R \cup DO \cup CO \mapsto V \cup S \cup \hat{I}$ and $t_a : \{n\} \mapsto (V \cup S) - \{\#\}$. Transitions are indicated as $s_1 \xrightarrow{t} s_2$, where $t(s) = s_1$ and $t(n) = s_2$. t^+ denotes a finite sequence of transitions, $t_0, t_1, \ldots, t_k$, such that for all $0 < i < k$, $t_i(n) = t_{i+1}(s)$, and $t^+(x) = t_k(t_{k-1}(\ldots(t_0(x))\ldots))$.

The *care* set for a transition denotes the set of registers, ports, predicates, inputs and state that do not have don't care values according to the assignment function t. $care_t = \{p \mid t(p) \neq \#\}$. For a sequence of transitions, the care set is defined as $care_{t^+} = \{p \mid t \in \{t^+\}\ .\ \exists t(p) \neq \#\}$.

A behavior table description of Hunt's verified FM9001 [10] processor is shown in Table 1. We will use this description as an example to illustrate our methodology.

3 Control and Datapath

The functional units (arithmetic/logic units, registers, switches) in a system and the interconnections between them are called the datapath of a system. The control determines the state, external and internal conditions that select the actions performed by the datapath. Indirection blurs the distinction between control and datapath in behavior tables, by providing a decision mechanism in the datapath.

We will define relations over machines based on relations over their states and transitions. A sequence of transitions *includes* another ($t_1^+ \preceq t_2^+$), if all decision conditions tested in one sequence of transitions are tested in the other, and all registers/ports with non don't care values at the end of one sequence of actions must have an equivalent value on the corresponding register/port at the end of the other sequence of actions.

Definition 3.1: A maximal relation over states ($\mathcal{S} \subseteq S_1 \times S_2$), is a *simulation* relation if $s_1 \sqsubset s_2$ implies :

$$\forall s_1 \xrightarrow{t_1^+} s_1'\ .\ \exists s_2 \xrightarrow{t_2^+} s_2'\ .\ t_1^+ \preceq t_2^+\ \wedge\ s_1' \sqsubset s_2'$$

where S_1, S_2 are sets of states.

Definition 3.2: The control and datapath of a machine M_1 are *implementated by* the control and datapath of machine M_2 ($M_1 \sqsubseteq M_2$) iff,

$$\forall s_1 \in S_1\ .\ \exists s_2 \in S_2\ .\ s_1 \sqsubset s_2$$

3.1 Transformations

Transformations that preserve the implementation relation are used to derive a realization from a specification. We will now illustrate a few transformations on the behavior table description of the FM9001.

In the decision section of the behavior table (Table 1), we generalize the functions (mode-a ins) and (mode-b ins) to (mode op ins) by adding a register op in the action section. The predicate columns (reg-direct-p (mode op ins)) can then be merged because they are *conflict-free* (defined in section 5). Similarly, both the (pre-dec-p ...) columns and both the (post-inc ...) columns can be merged. These transformations reduce the decision columns from ten to seven. The functions (rn-a ins), and (rn-b ins) in the action section (Table 1) can also be generalized using the op register.

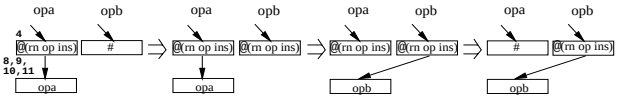

Figure 1: Datapath Transformations

Figure 1 shows the transformations on data-flow diagrams for registers opa and opb after transition 4. The value from the column opa is copied over the don't care value in opb for transitions 4, 5, 6, 7. We then change the values in column opa for transitions 8, 9, 10, 11, from opa to opb. This is a valid transformation because the values in opa and opb are always the same in the state op-b. The values in opa in the state op-b are "dead", and are replaced by don't cares in transitions 3, 4, 5, 6.

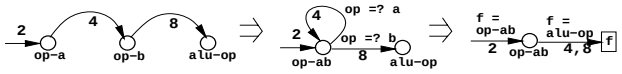

Figure 2: Control Transformations

We now consider transformations involving indirection. Indirection can be used to model a portion of the control in the datapath. A bounded indirect reference in the datapath can be transformed into a set of transitions, each with a decision for a possible reference.

As shown in Figure 2 we merge the states op-a and op-b into a single state op-ab, and use the value in register op to decide on which transition to take. Consider transitions 4 and 8, which differ only in the next-state column. A register future-state (f in Figure 2) is added, which precomputes the next state in all transitions leading to op-a, and gets the value alu-op in transitions 4 and 8. The next-state values for transitions 4, 8 are then replaced by future-state for state indirection. These two transitions can now be merged. Similarly, transitions 5 and 9, 6 and 10, and 7 and 11 can also be merged resulting in reduction of one state and four transitions.

As we have seen here, control and datapath transformations go hand in hand. This reinforces our main thesis, that a design representation which unifies different facets of a system in a single formal model can be used to address unexplored aspects of system design. Other control and datapath transformations are described in [11, 12].

4 Sequential Decomposition

Sequential decomposition is a generalization of *factorization*[9] to include protocols between components of a system. A behavior table can be decomposed into interacting behavior tables using a protocol specification with non-trivial control synchronization and data transfer interactions between components. This is different from classical FSM decomposition [13], which assumes tightly-coupled sub-machines that can share state and input information.

We use *Interface specification language (ISL)* to specify the interaction of a component with its environment over input/output ports [14]. ISL can be used to formalize timing diagrams for a component based on a reference clock speed. A specification in ISL constructs a machine with unique start and final states.

The *complement* of a machine describes the protocol behavior of its environment. Decomposition is accomplished by embedding the machine constructed by a protocol definition as a "stub" into one side of the protocol interface, and an implementation of an interaction path of its complement as the "stub" on the other side of the protocol interface.

For every output(input) port in a machine, a new input(output) port is created in its complement. A machine has the same set of states as its complement. In every transition, the complement machine has the same value on the corresponding ports.

Each path from the reset state to the final state in the complement machine represents a valid sequence of interactions to complete a protocol. A machine can interact with an implementation of any interaction path of its complement machine. The appropriate *path implementation* of the complement of a decomposed component is embedded into the system description.

The path implementation relation over machines is defined in terms of a maximal path simulation relation over their states. $s_1 \sqsubset_{\mathrm{p}} s_2$ implies that if there is a transition from s_1, then some transition from s_1 corresponds to a transition from s_2, and they lead to states in the same relation. For all transitions with active control inputs from s_1, there is a corresponding transition from s_2. Also, if s_1 is a wait state for a control input, then s_2 must also be a wait state for the same control input, or s_2 must lead to a wait state for the same control input [14].

Definition 4.3: A machine M_1 is *path implemented* by machine M_2 ($M_1 \sqsubseteq_{\mathrm{p}} M_2$) if, $r_1 \sqsubset_{\mathrm{p}} r_2 \ \wedge \ f_1 \sqsubset_{\mathrm{p}} f_2$, where r_1, r_2 are the unique start states, and f_1, f_2 are the unique final states of M_1, M_2.

We will use *sequential decomposition* to factor a memory object from the FM9001 description, based on the protocol specification of a memory sub-system, written below in ISL syntax:

$$\mathsf{Mem}(\mathsf{strobe}, \mathsf{RW}, \mathsf{ADDR}, \overline{\mathsf{dtack}}, \overline{\mathsf{DIN}}, \mathsf{DOUT}) \triangleq$$
$$[;\, \mathsf{strobe}, \mathsf{RW}/T : \mathsf{ADDR}/V_{\mathrm{addr}} \ \mathit{until}\ \overline{\mathsf{dtack}} : \overline{\mathsf{DIN}}/V_{\mathrm{read}}$$
$$[\!]\ ;\, \mathsf{strobe}, \mathsf{RW}/F : \mathsf{ADDR}/V_{\mathrm{addr}}, \mathsf{DOUT}/V_{\mathrm{write}} \ \mathit{until}\ \overline{\mathsf{dtack}}]^*$$

The state diagram for the memory is shown in Figure 3. $\overline{\mathsf{Mem}}$ is the complement of Mem. *Path implementations* of $\overline{\mathsf{Mem}}$ corresponding to Read and Write memory operations are embedded into the processor description in place of the transitions with read (Figure 4) and write operations. The ports in $\overline{\mathsf{Mem}}$ as added as columns in the behavior table. All read operations in the table are replaced by the port DIN, with appropriate values on $\overline{\mathsf{ADDR}}$ and $\overline{\mathsf{strobe}}$ in the embedded sequence of transitions. This assures us that the processor can interact properly with the memory.

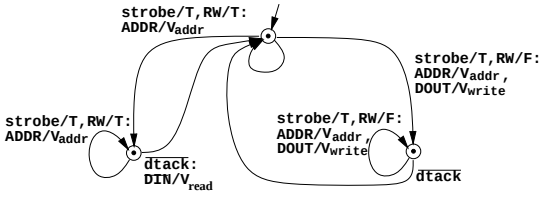

Figure 3: Memory Sub-system Protocol Specification

Our methodology is powerful enough to derive the memory controller for a DRAM memory system, with a protocol specification from DRAM timing diagrams, (reported in [15]).

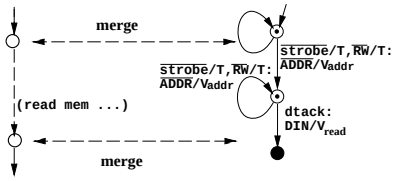

Figure 4: Embedding Path Implementation for Read

5 Data Expansion

Data expansion enables a designer to reason at the abstract symbolic level before assigning representations to functional units and values in a system. The organization of the behavior tables helps in visualizing values in different levels of abstraction in a system.

Transforming the data representation of a machine $M = \langle T, s, n, P, R, C, S, \hat{I}, V \rangle$ creates a new machine $M' = \langle T', \vec{s}, \vec{n}, P', R', C', S', \hat{I}', V' \rangle$. We define a representation function, that maps columns in M to a partition of columns in M'. A *type* is a finite domain for representation of values. $\mathcal{R} : \{s\} \cup \{n\} \cup P \cup R \cup C \times \ type \ \mapsto \{\vec{s}\} \cup \{\vec{n}\} \cup [P'] \cup [R'] \cup [C']$. We also define a function that maps values in one data type to vectors of values in another, $\mathcal{V} : V \cup S \cup \hat{I} \times \ type \ \mapsto V'^* \cup S'^* \cup \hat{I}'^*$.

For the data representation changes to be valid, the representation function $\mathcal{R}$ must be onto, and it must also be one-to-one or *conflict-free* [16]. We will assume that $\mathcal{R}$ is onto, and we define the *conflict-free* condition. If two or more columns are mapped on to the same column, then they must be *conflict-free*.

Definition 5.4: A data representation function $\mathcal{R}$ is *conflict-free* iff :

$$\forall p' \in P' \cup R' \cup C' \,.\, \exists p_1, p_2 \in P \cup R \cup C \,.$$
$$p' \in \mathcal{R}(p_1) \cap \mathcal{R}(p_2) \Rightarrow \forall t \in T \,.\, t(p_1) = \# \vee t(p_2) = \#$$

The above conditions are *sufficient* to state that $\mathcal{R}$ is valid and can be used to transform the machine M.

Definition 5.5: A machine *represents* another ($M' \sqsubseteq_{\mathcal{R}} M$) if each column in M is transformed using $\mathcal{R}, \mathcal{V}$ and a *type*, to a vector of columns in M'.

As an example, consider the change of representation for the column ins from symbolic to 32-bit boolean in the FM9001 (Figure 5). The only non don't care values in the column in Table 1, are ins and (read (@pc) mem). Assuming that (read (@pc) mem) has been replaced by DIN by memory decomposition, changing the representation of ins, will introduce a coercion on DIN from symbolic to 32-bit boolean. This in turn will generate a side-condition that the port $\overline{\text{DIN}}$ in Mem, which is connected to DIN, is also of type 32-bit boolean.

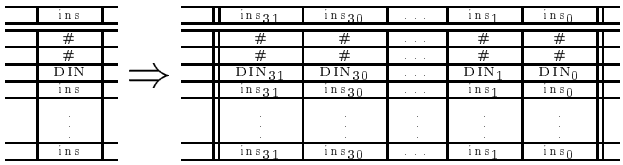

Figure 5: Data Abstraction Transformation

6 Conclusion

In this paper, we presented behavior tables as a unified representation for transformational system synthesis. Behavior tables enable a system designer to specify a system at a high-level of abstraction, and provide a design environment that can be used to transform the specification into an implementation of interacting sequential components at a lower level of abstraction. We introduced the use of indirection for system specification and synthesis. Using the transformations described here, the FM9001 description was reduced from five states, fifteen transitions, and ten predicates, to four states, eleven transitions, seven predicates, with a new register, and DRAM memory interface.

References

[1] H. Trickey, "Flamel: A high-level hardware compiler," in *TCAD 1987*, pp. 259–269, IEEE, Mar. 1987.

[2] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn, "The system architect's workbench," in *Proceedings of 25th DAC*, 1988.

[3] R. Jain, K. Küçükçakar, M. J. Mlinar, and A. C. Parker, "Experience with the ADAM synthesis system," in *Proceedings of 26th DAC*, 1989.

	present state	(nlist-p oracle)	(a-imm-p ins)	(reg-dir-p (mode-a ins))	(pre-dec-p (mode-a ins))	(post-inc-p (mode-a ins))	(reg-dir-p (mode-b ins))	(pre-dec-p (mode-b ins))	(post-inc-p (mode-b ins))	(store -resultp
0	intr	1	#	#	#	#	#	#	#	#
1	intr	0	#	#	#	#	#	#	#	#
2	fetch	#	#	#	#	#	#	#	#	#
3	op-a	#	1	0	0	0	#	#	#	#
4	op-a	#	0	1	0	0	#	#	#	#
5	op-a	#	0	0	1	0	#	#	#	#
6	op-a	#	0	0	0	1	#	#	#	#
7	op-a	#	0	0	0	0	#	#	#	#
8	op-b	#	#	#	#	#	1	0	0	#
9	op-b	#	#	#	#	#	0	1	0	#
10	op-b	#	#	#	#	#	0	0	1	#
11	op-b	#	#	#	#	#	0	0	0	#
12	alu-op	#	#	#	#	#	1	#	#	1
13	alu-op	#	#	#	#	#	0	#	#	1
14	alu-op	#	#	#	#	#	#	#	#	0

	next -state	regs	flags	mem	pc	ins	opa	opb	b-addr	oracle
0	intr	regs	flags	mem	#	#	#	#	#	oracle
1	fetch	regs	flags	mem	(car oracle)	#	#	#	#	(cdr oracle)
2	op-a	(ur regs pc (inc (@pc)))	flags	mem	pc	(read (@pc) mem)	#	#	#	oracle
3	op-b	regs	flags	mem	pc	ins	(extend (a-imm ins) 32)	#	#	oracle
4	op-b	regs	flags	mem	pc	ins	(@(rn-a ins))	#	#	oracle
5	op-b	(ur regs (rn-a ins) (dec (@ (rn-a ins))))	flags	mem	pc	ins	(read (dec (@(rn-a ins))) mem)	#	#	oracle
6	op-b	(ur regs (rn-a ins) (inc (@ (rn-a ins))))	flags	mem	pc	ins	(read (inc (@(rn-a ins))) mem)	#	#	oracle
7	op-b	regs	flags	mem	pc	ins	(read (@(rn-a ins)) mem)	#	#	oracle
8	alu-op	regs	flags	mem	pc	ins	opa	(@(rn-b ins))	#	oracle
9	alu-op	(ur regs (rn-b ins) (dec (@(rn-b ins))))	flags	mem	pc	ins	opa	(read (dec (@(rn-b ins))) mem)	(dec @(rn-b ins))	oracle
10	alu-op	(ur regs (rn-b ins) (inc (@ (rn-b ins))))	flags	mem	pc	ins	opa	(read (inc (@(rn-b ins))) mem)	(@(rn-b ins))	oracle
11	alu-op	regs	flags	mem	pc	ins	opa	(read (@(rn-b ins)) mem)	(@(rn-b ins))	oracle
12	intr	(ur regs (rn-b ins) (do-op ...))	(uf flags ins opa opb)	mem	pc	ins	opa	opb	b-addr	oracle
13	intr	regs	(uf flags ins opa opb)	(write b-addr mem (do-op flags opa opb ins))	pc	ins	opa	opb	b-addr	oracle
14	intr	regs	(uf flags ins opa opb)	mem	pc	ins	opa	opb	b-addr	oracle

Table 1: FM9001 Behavior Table Specification - Decision and Action Sections

[4] Z. Peng, *A Formal Methodology for Automated Synthesis of VLSI Systems*. PhD thesis, Linköping University, Sweden, 1987.

[5] M. R. K. Patel, "A design representation for high level synthesis," in *Proceedings of EDAC*, 1990.

[6] F. Vahid and D. D. Gajski, "Specification partitioning for system design," in *Proceedings of 29th DAC*, 1992.

[7] K. Küçükçakar and A. C. Parker, "CHOP: A constraint-driven system-level partitioner," in *Proceedings of 28th DAC*, 1991.

[8] S. D. Johnson, *Synthesis of Digital Designs from Recursion Equations*. Cambridge: MIT Press, 1984.

[9] S. D. Johnson, "Manipulating logical organization with system factorizations," in *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, *LNCS* 408, Springer, 1989.

[10] W. A. Hunt, "A formal HDL and its use in the FM9001 verification," in *Mechanized Reasoning in Hardware Design*, Prentice-Hall, 1992.

[11] B. Bose and S. D. Johnson, "DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation," in *Proceedings of CHARME*, Springer, 1993. LNCS 683.

[12] B. Bose, "DDD - A Transformation system for Digital Design Derivation," Tech. Rep. 331, Indiana University, Computer Science Department, May 1991.

[13] S. Devadas and A. R. Newton, "Decomposition and factorization of sequential finite state machines," *TCAD 1989*, vol. 8, pp. 1206–1217, Nov. 1989.

[14] K. Rath and S. D. Johnson, "Toward a basis for protocol specification and process decomposition," in *Proceedings of CHDL*, Elsevier, Apr. 1993.

[15] K. Rath, B. Bose, and S. D. Johnson, "Derivation of a DRAM memory interface by sequential decomposition," in *Proceedings of ICCD*, IEEE, 1993.

[16] Z. Zhu and S. D. Johnson, "An algebraic framework for data abstraction in hardware description," in *Designing Correct Circuits*, Springer, 1990.